

# L3/Filter Interface Specification

Amber Boehnlein      Dan Claes      Dave Cutts  
Jan Hoftun      Moacyr Souza      Gordon Watts

July 2, 1997

## **Abstract**

The interface between Level 3 filters and tools and the Level 3 framework is described in detail, including a quick reference section.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Environment</b>	<b>3</b>
2.1	General Rules . . . . .	4
2.2	Framework and <code>script_runner</code> . . . . .	5
2.3	Event Model . . . . .	6
2.4	RCP Interface . . . . .	6
<b>3</b>	<b>Trigger Information Interface</b>	<b>7</b>
3.1	Trigger Scripts . . . . .	7
3.2	Algorithm Parameters . . . . .	8
3.3	The Trigger Decision . . . . .	8
3.4	Other Databases . . . . .	8
<b>4</b>	<b>Special Filter Tools</b>	<b>9</b>
4.1	The Output Filter Tool . . . . .	9
<b>5</b>	<b>Monitor Information</b>	<b>9</b>
5.1	Monitor Utilities . . . . .	9
5.1.1	Timing filters and tools . . . . .	9
5.1.2	Histograms . . . . .	9
5.2	End Of Run Summary Data . . . . .	10
5.3	Event-by-Event . . . . .	10
5.4	Online Monitor Data . . . . .	10
<b>6</b>	<b>Errors and Log Files</b>	<b>10</b>
6.1	Log Files . . . . .	11
6.2	Errors . . . . .	11
<b>7</b>	<b>Interrupting Event Processing</b>	<b>11</b>
7.1	Gong API . . . . .	12
7.2	Memory API . . . . .	13
<b>8</b>	<b>Crashes in Filter Code</b>	<b>13</b>
<b>9</b>	<b>Framework Errors</b>	<b>14</b>
9.1	Corrupt Level 3 Data . . . . .	14

<b>10 Code and Library Management</b>	<b>15</b>
10.1 The 3AM problem . . . . .	15
10.2 Physical Design . . . . .	15
<b>11 Quick Reference</b>	<b>15</b>
<b>12 Outstanding Issues</b>	<b>17</b>

## 1 Introduction

How to read: details and quick ref section

This is expected to be a living document till it is done...

Where the image sits in the scheme of things and generic data flow. The onion diagram that we've been passing around. A generic, quick, list of the sorts of information that must be passed between the outside world.

A line or two about script runner, define filters and tools, resets (will have to be in RCP defs below).

## 2 Environment

The Level 3 trigger operates in two environments: online in a Level 3 trigger processing node running Windows NT as its OS and offline, in the form of a simulator, running on many platforms.

The online system is the most demanding: there can be no text i/o to the terminal for example and the algorithms must be very fast! The simulation environment is a bit more lax, but the simulator must work in the standard DØ offline reconstruction programs. This may mean extra software packages to help process the output of the Level 3 trigger.

The implementation assumes that the trigger decision is made by a single software package.<sup>1</sup> The Level 3 framework passes control to this package with a standard DØ event pointer and expects in return a pass/fail decision. The simulator operates in a similar way, with the same Level 3 code added as a package to the offline processing framework.

This section describes the framework and the event model and how it relates to the Level 3 framework and simulator.

---

<sup>1</sup>This could be a super package, which contains several packages.

Table 1: The memory allocation in a typical Level 3 node as it is imagined for Run 2.

Note that the name `script_runner` has been settled on as the name of the package that implements the trigger.

## 2.1 General Rules

The trigger program must operate in a quasi-realtime environment. This means there are time and memory restrictions put on the program that are not present in offline processing, among other things.

The Level 3 system is expected to process 1000 events a second at the outset of Run 2. This translates to between 25 and 100 milliseconds per event, depending upon the node configuration. If once in a while an event takes too long, it must be terminated (see Section 7.1).

The same is true for memory. It is expected the memory budget for the Level 3 processing nodes will look something like Table 1, at least initially. While these are Windows NT systems that have paging disks attached, in no way can a trigger or other stuff cause a page fault: it takes too long. Thus, all the data structures, geometry databases, calibration databases, code, etc. must be less than xxxxMBytes. There are provisions for watching over memory allocation and terminating an event's processing should the memory demands exceed what is available (see Section 7.2).

There are a number of generic restrictions as well. For example, the trigger tools and scripts shouldn't try to access something off site of the machine on an event-by-event basis, or event at run initialization time. For two reasons: first network access is slow and second it is likely that the Level 3 trigger processors will be on their own isolated network link and connectivity to the outside world restrained. Besides, if you have Level 3 access the data it can cache it locally on each node making getting the data much faster.

Nor should any low level OS calls be made, either UNIX (any supported flavor) or NT. The trigger code must run on both platforms, and, so, cannot program to one OS or the other. If there is a service required contact the Level 3 group to have it provided in some sort of platform independent way.

Actions: create, destroy, process\_event. Specify RCP parameters to the creator!

Table 2: The actions the `script_runner` object must respond to, weather they are from the Level 3 or offline (simulation) framework.

## 2.2 Framework and `script_runner`

The standard DØ framework is used in Level 3 code. The document xxx, DØ node xxx describes it fully. Here the parts of the framework key to the Level 3 operation are highlighted.

The framework does not exist *per se* in the online environment. Rather, the standard framework package API is used for `script_runner`. This, of course, implies that `script_runner` is an object that inherits from the `d0_package` object.

In the simulation the `script_runner` exists as a full class package.

In both the online and offline the messages sent to the `script_runner` package are the same. The most important actions shown in Table 2.

When the script runner object is created its constructor is passed an RCP object specifying the trigger scripts it should run. See Section 3.1 for a detailed description of the RCP object. Once parsed and verified the constructor returns. If an error occurs during construction of the `script_runner` object then xxxxx. As is specified in the Framework document, the RCP object is inaccessible after the `script_runner` object returns.

When the L3 framework is being torn down or a offline simulation has finished the destructor for the created `script_runner` object is invoked. All allocated memory should be cleaned up at this point. There is no provision for information to be returned at this point. An exception during destruction is ignored and L3 framework will continue unabated. ???xxxx???

Note that the specification of the DØ online says that a single instance of `script_runner` could exist across more than a single run. It is also possible for `script_runner` to be given events from more than one run during a simulation. However, `script_runner` should not change its triggering behavior after construction is finished.

Between construction and destruction, `script_runner` does its real work of filtering events. The messages that `script_runner` must respond to are listed in brief in Table 3. Detailed actions and expected actions are listed in the sections listed in the table. In brief, the first message `script_runner`

all the messages and sections where they are described in detail

Table 3: The messages the `script_runner` package must respond to along with a short description and the Section in this document where the behavior of the `script_runner` package response is described in detail.

receives is a `begin_run`. All internal run-state variables should be cleared at this point. It will then receive a number of `process_event` messages along with a standard DØ event object (see Section 2.3). It should return a pass/fail decision and modify the event object as it sees fit. If the event is passed the Level 3 framework will pass the event up to the host system. In the simulation the event will be passed to the next module. Finally, the framework will send an `end_run` message to `script_runner`, again with a standard event object. The `script_runner` package should add a summary chunk (see Section 5.2) that contains end-of-run information. The `script_runner` package should then be ready for the next `begin_run` message.

## 2.3 Event Model

The standard DØ event model is used in Level 3 code. The document xxx, DØ node xxx describes it fully.

There are no special considerations for using the event model in Level 3 (???), except to note that xxxxx.

The Level 3 framework and offline simulator expect only on extra chunk, the data summary chunk that should be added to an event at the end of run. See Section 5.2 for a complete description.

## 2.4 RCP Interface

The standard DØ RCP interface is used in Level 3 code. The document xxx, DØ node xxx describes it fully.

The RCP interface is used heavily during trigger definition time: the complete trigger specification for the `script_runner` package is passed in via a large RCP object. See Section 3.1 for detailed information.

Figure 1: An example trigger filter script

### 3 Trigger Information Interface

The `script_runner` package must access a considerable amount of information to complete its task. From the trigger script definitions all the way down to calibration databases. This section attempts to describe how these various databases will be accessed online, as well as list the things that are offline in this restricted environment. Of course, full access is available during simulation as the environment is the offline, however, since the code is expected to be the same offline and online, it is not legal to access the databases during `script_runner` execution. A tool will be provided to allow the filter tool writer to assure their filter or tool does not access any of the disallowed databases or information sources.

#### 3.1 Trigger Scripts

The scripts are passed to the script runner package using the standard RCP protocol. The RCP bank will be manufactured at download time by the Level 3 system and passed to the script object's constructor. For a detailed description of the RCP system, refer to DØNOTE xxxx.

The trigger script is accessible with the key `TRIGGER_SCRIPT`, and it is another RCP object. Table 4 describes the layout of the trigger script RCP object. It is imagined that the script runner will parse the trigger in the following way:

1. Use the key `NUMBER_DEFINED_BITS` to fetch the number of defined trigger bits that script runner should use.
2. Use the key `DEFINED_BITS` to fetch an integer array of length `NUMBER_DEFINED_BITS`, each entry is a Level 3 trigger bit number.
3. For each defined trigger bit, fetch the RCP object associated with the key `BIT_DEFINITION_n` where `n` is a trigger bit number from the array `DEFINED_BITS`. Table 5 shows the contents of each trigger bit RCP object. Figure 1 shows an example trigger script.

Table 4: The contents of the trigger script RCP object. This top level RCP object contains the list of defined bits as well as another RCP object for each defined bit.

RCP Key	Type	Definition
L2_BIT	Integer	Level 2 bit number this L3 bit hangs on
SCRIPT	String	Level 3 trigger script

## 3.2 Algorithm Parameters

Filter tools need two sources of input to work: a refset definition and a set of tool parameters. The refset describes things like which side cone jet to create, what is the minimum  $E_T$  of the found jets. The parameters are used by the author of the tool to tune the algorithm.

The regsets come in as described above. The algorithm parameters come in as xxx

## 3.3 The Trigger Decision

For an event to be properly written out to the host system, two things must happen. Script Runner must return success to the `process_event` message and `event_done` message, and it must invoke the special filter tool, the output filter tool (see Section 4). If both conditions are satisfied, the event will be written to the host system.

## 3.4 Other Databases

I imagine this will be the same as offline (please, pretty please). And there will be some large complex ones unavailable, but there may also be special reduced versions of some of them... in that case we have to make sure that `script_runner` accesses those during the simulation, but other components of the offline access the full blown databases.

Table 5: The contents of each filter bit's RCP object.

## 4 Special Filter Tools

This section describes tools that are really part of the framework. For example, the output tool that can be executed cause an event to be written out to the host. It may also be part of the preceding section (?).

The prescale tool isn't discussed here as it has not direct connection with the framework. Rather it is a simple filter that passes one of n.

### 4.1 The Output Filter Tool

Takes a stream name as an argument. May encode in event in standard way so data logger can route the event appropriately.

## 5 Monitor Information

The filter scripts as well as other components of trigger will want to save data at the end of run. This data includes histograms as well as counters and averages. The trick, in this environment, is that there are an unknown number of other nodes running the same information that must be matched and collated.

How do you deal with more than one trigger list in a node?

This section describes how to save summary data that is collected at the end of a run and some common utilities that are used during monitoring.

### 5.1 Monitor Utilities

#### 5.1.1 Timing filters and tools

The standard DØ timing classes can be used to time the filter tools. Fritz.

#### 5.1.2 Histograms

Use the CD's histoscope, pulled a part (as they did for VXWORKS – I have names if the API is up to the job).

Figure 2: API for the summary data chunk. This chunk is expected at end of run to contain the data that is a summary of this filters running.

## 5.2 End Of Run Summary Data

When the `end_of_run` message is sent to the script runner all summary data should be assembled and stored in a summary chunk in the event.

The summary chunk has the interface specified in Figure 2.

In the simulation there would be a downstream module that would parse the summary chunk information and produce a text output for it.

Other way to do it is to create summary objects that do things like average, etc. Then at end run automatically loop over them all and extract their information. How would this work in the simulation?

## 5.3 Event-by-Event

Event-by-Event data is not collected, per say, by the L3 trigger. If a significant amount of data needs to be written out, it must be send along the usual data stream path. That is, out to the host system and into a compressed (?) stream itself. The design of the L3 framework is flexible enough to handle one path of large sized events and a second path with small, reduced data set events.

## 5.4 Online Monitor Data

The online system needs to keep track of the Level 3 trigger/DAQ status, and, in particular, the nodes that are working. Any information that is provided to the online system is provided on a as-need basis. That is, the online system must explicitly request the information.

Thus, script runner must by async to handle these requests. Blah Blah.

# 6 Errors and Log Files

The Level 3 filter and framework code are expected to generate exceptions, as well as log entries. Note that during normal running no errors or log file messages should be generated as each message requires a certain amount of

overhead: errors must be sent over the network to the alarm server and as well as written to a local log file, for example.

## 6.1 Log Files

The Level 3 filter and tool code should use the standard DØ log file routines. Detailed information on the DØ logfile library can be found in the DØNOTE xxxx. The most common calls are repeated in the Quick Reference Section, in Section 11.

## 6.2 Errors

The Level 3 filter and tool code should use the standard DØ error routines. Detailed information on the DØ error library can be found in the DØNOTE xxxx. The most common calls are repeated in the Quick Reference Section, on page 11.

Special care must be taken when sending error messages. The Level 3 framework will report all error messages sent to it as follows:

- All errors above a priority of xxx will be reported to the alarm server and thus to the global error logs
- If the same error occurs more than 20 times in a node, reporting to the alarm server will be suppressed.
- All errors will be written to the local node log file

In normal running there should be no errors or other output. The reason is that everytime there is an error it must be recorded to the disk or across a network; a bit of a waste.

## 7 Interrupting Event Processing

There are several reasons to voluntarily terminate event processing. For example, if an event is taking too long or its memory requirements are outrageous, it may be that the event should be terminated and passed on to the host.

There are several issues that must be addressed here. First, there is the matter of event clean up. Since the event model specifies that we cannot

```

#include "l3_framework.h"

void force_gong (void);
void force_gong (void)
{
    l3_framework_util::interrupt_processing (
    err_error (ERINFO, "Event took too long"));
}

```

Figure 3: Example of a callback to interrupt trigger processing for an event that has run over time.

modify a chunk after adding it to the event, temporary results that are modified as the trigger runs can't be added to the event. The trigger bits that have fired are an example of this. However, due to the asynchronous nature of the interrupts being discussed here the code to add the chunk at the end of the event will never be processed.

Also, it may be desirable to add temporary result chunks for debugging purposes to the event that would not be added during normal execution of the trigger.

Both of these issues are addressed using a non-standard reconstructor message, `end_of_event`. This is invoked at the end of every event in the standard way. Code may then be written to hang off this message that will add information to the event. Care must be taken to account for both the possibility of multithreading and that the reconstructor object may not be in a totally consistent state. Section 8 describes what happens when a crash occurs during processing of the `end_of_event` message.

The third issue is that the script runner must be capable of telling the framework to interrupt processing of the event. For this a callback is provided. As arguments it takes only an error object which should describe the reasons for the processing interruption. The error will be reported via the standard mechanism. Figure 3 shows an example, and Figure 4 the API.

## 7.1 Gong API

Occasionally an event will take too long to complete trigger processing. This isn't always caused by a bug. For example, in Run 1 if an event had a huge

Routine	<code>interrupt_processing</code>
Arguments	<code>interrupt_processing (err_error reason_for_interruption)</code>
<code>reason_for_interruption</code>	Error object describing why the fellow has failed.
Notes	No detstructors are called for objects on the stack, so watch for possible

Figure 4: The API for interrupting filter processing.

Figure 5: Example code for causing a gong after 150 milliseconds have elapsed. The sample routine `force_gong` is shown in Figure 4.

number of muon hits the muon tracking could take 10s of seconds instead of the standard milliseconds.

In Run 1 events like these were cut off after a certain amount of time to prevent them from taking over the farm. Once interrupted the events were marked and shipped off to the host with the filter bits already processed set and the unprocessed bits artificially set on.

To accomplish this functionality in the Run 2 filter framework a timer should be created by the script runner and started before filter processing commences. When the timer fires it should invoke a routine similar to `force_gong`, as shown in Figure 3. The timing objects described in Section 5.1.1. An example is shown in Figure 5.

## 7.2 Memory API

Sometimes an event will use an inordinate amount of memory and it is desirable to terminate the processing before it runs completely amok.

Detecting this a little more difficult, and there are two possible approaches:

- thread to watch memory
- overriding the global memory allocator.

## 8 Crashes in Filter Code

If the filter code (or the framework) causes a operating system exception, divide by zero or attempts to write to write-protected memory, a crash will

occur. Control will be handed to a special part of the framework which will do one of the following, depending upon the running mode:

1. In debug mode: Start the debugger and inform the DAQ shifter so that the system might be debugged.
2. In Auto mode: the `end_of_event` message will be passed to script runner. If the message completes, the event object will be packaged up and written to the host system. If another crash occurs during the processing of this message, then only the raw data will be shipped off to the host system.

## 9 Framework Errors

Sometimes the framework goes south before the filter tools ever get executed.

### 9.1 Corrupt Level 3 Data

The Level 3 framework will perform a basic check of the data before handing it off to the script runner. This includes things like a checksum. Events that fail the checksum can have one of several things happen to them, depending upon the mode:

- The event is discarded, and the filter framework moves on to the next one.
- Debugger invoked and the DAQ shifter is notified.
- The raw data is shipped upto the host system and the filter framework moves onto the next event.

If a node fails these checks on more than 50 events in a row, that node will be disabled until the next begin run (an error message will be sent around). The number 50 is a settable parameter.

## 10 Code and Library Management

Issues relating to keeping versions constant as they move into the L3 NT environment. I'm not sure how much will be in this section. Perhaps it is nothing more than a list of requirements that the L3 group must fill. Version number info in here too?

Simulation. Getting an "official" library to link in instead of the most recent offline...

### 10.1 The 3AM problem

10 - 15 minutes from change to new L3 Exe. Auto build for NT system.

### 10.2 Physical Design

The physical design of the software must be layed out to facilitate not only fast and easy loading in the online system but also use as a package for trigger simulation in the offline system. Figure 6 depicts the relationship and dependencies of the major components of the online trigger and simulator.

The script runner and filter component are be built with only the standard DØ package interface exposed, and will require only a few L3 framework objects (not shown) and the event model library to function. None of the routines inside the tools are exposed for external use. xxx???

Because there must be common access to the data, both in online and offline the event model routines are common. This presents the problem that if the event model is updated in some fundamental way after the trigger is built that there may be access conflicts.

This design also allows for different versions of online routines to exist in the offline library, and have the trigger simulator use the online version and the offline use the offline version.

## 11 Quick Reference

This section describes the APIs used in this document with little or no explaining text. References to other documents or page numbers in this document are given if more information is needed.

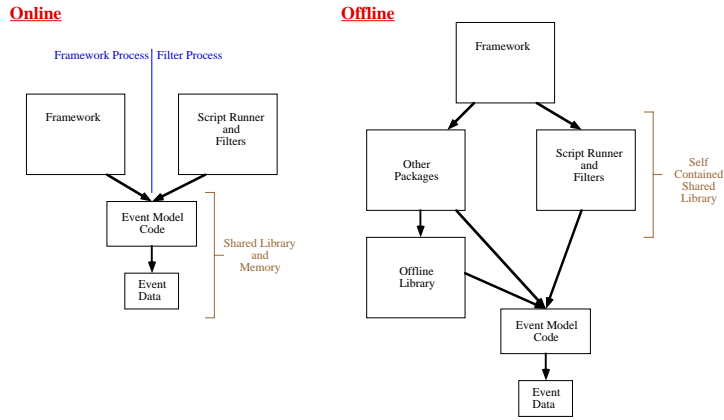


Figure 6: The physical layout of the online trigger software and the offline simulator. In both cases the script runner and tools exist as a separate code base, self contained to allow for the same code to be used offline and online even in the presents of different versions of online code present in a current release of the offline software.

## 12 Outstanding Issues

Mark and Pass runs

Anything about shadow in here right now?